50 Ways to Leak Your Data:
An Exploration of Apps' Circumvention of the Android Permissions System

Joel Reardon
University of Calgary
AppCensus, Inc.

Álvaro Feal
IMDEA Networks Institute
Universidad Carlos III de Madrid

Primal Wijesekera
U.C. Berkeley / ICSI

Amit Elazari Bar On
U.C. Berkeley

Narseo Vallina-Rodriguez
IMDEA Networks Institute / ICSI
AppCensus, Inc.

Serge Egelman
U.C. Berkeley / ICSI
AppCensus, Inc.

Abstract

Modern smartphone platforms implement permission-based models to protect access to sensitive data and system resources. However, apps can circumvent the permission model and gain access to protected data without user consent by using both covert and side channels. Side channels present in the implementation of the permission system allow apps to access protected data and system resources without permission; whereas covert channels enable communication between two colluding apps so that one app can share its permission-protected data with another app lacking those permissions. Both pose threats to user privacy.

In this work, we make use of our infrastructure that runs hundreds of thousands of apps in an instrumented environment. This testing environment includes mechanisms to monitor apps' runtime behaviour and network traffic. We look for evidence of side and covert channels being used in practice by searching for sensitive data being sent over the network for which the sending app did not have permissions to access it. We then reverse engineer the apps and third-party libraries responsible for this behaviour to determine how the unauthorized access occurred. We also use software fingerprinting methods to measure the static prevalence of the technique that we discover among other apps in our corpus.

Using this testing environment and method, we uncovered a number of side and covert channels in active use by hundreds of popular apps and third-party SDKs to obtain unauthorized access to both unique identifiers as well as geolocation data. We have responsibly disclosed our findings to Google and have received a bug bounty for our work.

1   Introduction

Smartphones are used as general-purpose computers and therefore have access to a great deal of sensitive system resources (*e.g.*, sensors such as the camera, microphone, or GPS), private data from the end user (*e.g.*, user email or contacts list), and various persistent identifiers (*e.g.*, IMEI). It is crucial to protect this information from unauthorized access. Android, the most-popular mobile phone operating system [75], implements a permission-based system to regulate access to these sensitive resources by third-party applications. In this model, app developers must explicitly request *permission* to access sensitive resources in their Android Manifest file [5]. This model is supposed to give users control in deciding which apps can access which resources and information; in practice it does not address the issue completely [30, 86].

The Android operating system sandboxes user-space apps to prevent them from interacting arbitrarily with other running apps. Android implements isolation by assigning each app a separate *user ID* and further mandatory access controls are implemented using SELinux. Each running process of an app can be either code from the app itself or from SDK libraries embedded within the app; these SDKs can come from Android (*e.g.*, official Android support libraries) or from third-party providers. App developers integrate third-party libraries in their software for things like crash reporting, development support, analytics services, social-network integration, and advertising [16, 62]. By design, any third-party service bundled in an Android app inherits access to all permission-protected resources that the user grants to the app. In other words, if an app can access the user's location, then all third-party services embedded in that app can as well.

In practice, security mechanisms can often be circumvented; *side channels* and *covert channels* are two common techniques to circumvent a security mechanism. These channels occur when there is an alternate means to access the protected resource that is not audited by the security mechanism, thus leaving the resource unprotected. A *side channel* exposes a path to a resource that is outside the security mechanism; this can be because of a flaw in the design of the security mechanism or a flaw in the implementation of the design. A classic example of a side channel is that power usage of hardware when performing cryptographic operations can leak the particulars of a secret key [42]. As an example in the physical world, the frequency of pizza deliveries to government buildings may leak information about political crises [69].

A *covert channel* is a more deliberate and intentional effort between two cooperating entities so that one with access to some data provides it to the other entity without access to the data in violation of the security mechanism [43]. As an example, someone could execute an algorithm that alternates between high and low CPU load to pass a binary message to another party observing the CPU load.

The research community has previously explored the potential for covert channels in Android using local sockets and shared storage [49], as well as other unorthodox means, such as vibrations and accelerometer data to send and receive data between two coordinated apps [3]. Examples of side channels include using device sensors to infer the gender of the user [51] or uniquely identify the user [72]. More recently, researchers demonstrated a new permission-less device fingerprinting technique that allows tracking Android and iOS devices across the Internet by using factory-set sensor calibration details [90]. However, there has been little research in detecting and measuring at scale the prevalence of covert and side channels in apps that are available in the Google Play Store. Only isolated instances of malicious apps or libraries inferring users' locations from WiFi access points were reported, a side channel that was abused in practice and resulted in about a million dollar fine by regulators [82].

In fact, most of the existing literature is focused on understanding personal data collection using the system-supported access control mechanisms (*i.e.*, Android permissions). With increased regulatory attention to data privacy and issues surrounding user consent, we believe it is imperative to understand the effectiveness (and limitations) of the permission system and whether it is being circumvented as a preliminary step towards implementing effective defenses.

To this end, we extend the state of the art by developing methods to detect actual circumvention of the Android permission system, at scale in real apps by using a combination of dynamic and static analysis. We automatically executed over 88,000 Android apps in a heavily instrumented environment with capabilities to monitor apps' behaviours at the system and network level, including a TLS man-in-the-middle proxy. In short, we ran apps to see when permission-protected data was transmitted by the device, and scanned the apps to see which ones *should not* have been able to access the transmitted data due to a lack of granted permissions. We grouped our findings by *where* on the Internet *what* data type was sent, as this allows us to attribute the observations to the actual app developer or embedded third-party libraries. We then reverse engineered the responsible component to determine exactly how the data was accessed. Finally, we statically analyzed our entire dataset to measure the prevalence of the channel. We focus on a subset of the *dangerous* permissions that prevent apps from accessing location data and identifiers. Instead of imagining new channels, our work focuses on tracing evidence that suggests that side- and covert-channel abuse is occurring in practice.

We studied more than 88,000 apps across each category from the U.S. Google Play Store. We found a number of side and covert channels in active use, responsibly disclosed our findings to Google and the U.S. Federal Trade Commission (FTC), and received a bug bounty for our efforts.

In summary, the contributions of this work include:

- We designed a pipeline for automatically discovering vulnerabilities in the Android permissions system through a combination of dynamic and static analysis, in effect creating a scalable honeypot environment.
- We tested our pipeline on more than 88,000 apps and discovered a number of vulnerabilities, which we responsibly disclosed. These apps were downloaded from the U.S. Google Play Store and include popular apps from all categories. We further describe the vulnerabilities in detail, and measure the degree to which they are in active use, and thus pose a threat to users. We discovered covert and side channels used in the wild that compromise both users' location data and persistent identifiers.
- We discovered companies getting the MAC addresses of the connected WiFi base stations from the ARP cache. This can be used as a surrogate for location data. We found 5 apps exploiting this vulnerability and 5 with the pertinent code to do so.
- We discovered Unity obtaining the device MAC address using `ioctl` system calls. The MAC address can be used to uniquely identify the device. We found 42 apps exploiting this vulnerability and 12,408 apps with the pertinent code to do so.
- We also discovered that third-party libraries provided by two Chinese companies—Baidu and Salmonads—independently make use of the SD card as a covert channel, so that when an app can read the phone's IMEI, it stores it for other apps that cannot. We found 159 apps with the potential to exploit this covert channel and empirically found 13 apps doing so.
- We found one app that used picture metadata as a side channel to access precise location information despite not holding location permissions.

These deceptive practices allow developers to access users' private data without consent, undermining user privacy and giving rise to both legal and ethical concerns. Data protection legislation around the world—including the General Data Protection Regulation (GDPR) in Europe, the California Consumer Privacy Act (CCPA) and consumer protection laws, such as the Federal Trade Commission Act—enforce transparency on the data collection, processing, and sharing practices of mobile applications.

This paper is organized as follows: Section 2 gives more background information on the concepts discussed in the introduction. Section 3 describes our system to discover vulnerabilities in detail. Section 4 provides the results from our

study, including the side and covert channels we discovered and their prevalence in practice. Section 5 describes related work. Section 6 discusses their potential legal implications. Section 7 discusses limitations to our approach and concludes with future work.

2 Background

The Android permissions system has evolved over the years from an ask-on-install approach to an ask-on-first-use approach. While this change impacts when permissions are granted and how users can use contextual information to reason about the appropriateness of a permission request, the backend enforcement mechanisms have remained largely unchanged. We look at how the design and implementation of the permission model has been exploited by apps to bypass these protections.

2.1 Android Permissions

Android's permissions system is based on the security principle of *least privilege*. That is, an entity should only have the minimum capabilities it needs to perform its task. This standard design principle for security implies that if an app acts maliciously, the damage will be limited. Developers must declare the permissions that their apps need beforehand, and the user is given an opportunity to review them and decide whether to install the app. The Android platform, however, does not judge whether the set of requested permissions are all strictly necessary for the app to function. Developers are free to request more permissions than they actually need and users are expected to judge if they are reasonable.

The Android permission model has two important aspects: obtaining user consent before an app is able to access any of its requested permission-protected resources, and then ensuring that the app cannot access resources for which the user has not granted consent. There is a long line of work uncovering issues on how the permission model interacts with the user: users are inadequately informed about why apps need permissions at installation time, users misunderstand exactly what the purpose of different permissions are, and users lack context and transparency into how apps will ultimately use their granted permissions [24, 30, 78, 86]. While all of these are critical issues that need attention, the focus of our work is to understand how apps are circumventing system checks to verify that apps have been granted various permissions.

When an app requests a permission-protected resource, the resource manager (*e.g.*, `LocationManager`, `WiFiManager`, etc.) contacts the `ActivityServiceManager`, which is the *reference monitor* in Android. The resource request originates from the sandboxed app, and the final verification happens inside the Android platform code. The platform is a Java operating system that runs in system space and acts as an interface for a customized Linux kernel, though apps can interact with the kernel directly as well. For some permission-protected resources, such as network sockets, the reference monitor is the kernel, and the request for such resources bypasses the platform framework and directly contacts the kernel. Our work discusses how real-world apps circumvent these system checks placed in the kernel and the platform layers.

The Android permissions system serves an important purpose: to protect users' privacy and sensitive system resources from deceptive, malicious, and abusive actors. At the very least, if a user denies an app a permission, then that app should not be able to access data protected by that permission [24,81]. In practice, this is not always the case.

2.2 Circumvention

Apps can circumvent the Android permission model in different ways [3, 17, 49, 51, 52, 54, 70, 72, 74]. The use of covert and side channels, however, is particularly troublesome as their usage indicates deceptive practices that might mislead even diligent users, while underscoring a security vulnerability in the operating system. In fact, the United State's Federal Trade Commission (FTC) has fined mobile developers and third-party libraries for exploiting side channels: using the MAC address of the WiFi access point to infer the user's location [82]. Figure 1 illustrates the difference between covert and side channels and shows how an app that is denied permission by a security mechanism is able to still access that information.

Covert Channel A covert channel is a communication path between two parties (*e.g.*, two mobile apps) that allows them to transfer information that the relevant security enforcement mechanism deems the recipient unauthorized to receive [18]. For example, imagine that AliceApp has been granted permission through the Android API to access the phone's IMEI (a persistent identifier), but BobApp has been denied access to that same data. A covert channel is created when AliceApp legitimately reads the IMEI and then gives it to BobApp, even though BobApp has already been denied access to this same data when requesting it through the proper permission-protected Android APIs.

In the case of Android, different covert channels have been proposed to enable communication between apps. This includes exotic mediums such as ultrasonic audio beacons and vibrations [17, 26]. Apps can also communicate using an external network server to exchange information when no other opportunity exists. Our work, however, exposes that rudimentary covert channels, such as shared storage, are being used in practice at scale.

Side Channel A side channel is a communication path that allows a party to obtain privileged information without relevant permission checks occurring. This can be due to nonconventional unprivileged functions or features, as well as ersatz versions of the same information being available without
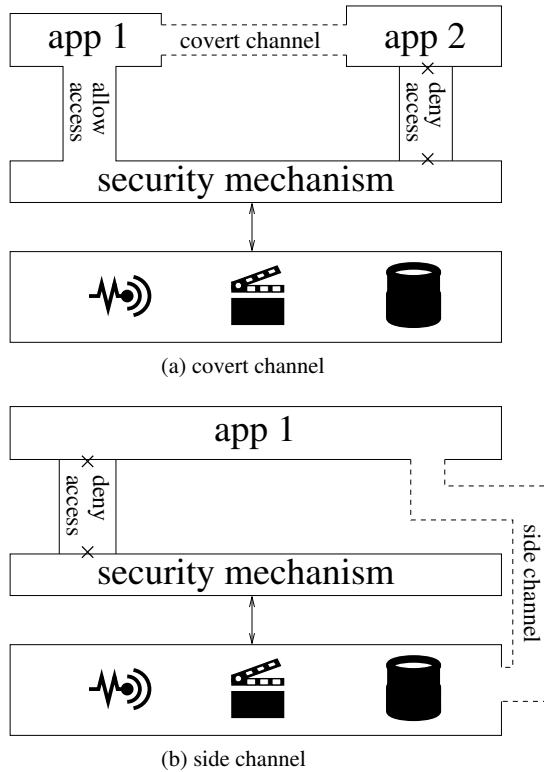
(a) covert channel

(b) side channel

Figure 1: Covert and side channels. (a) A security mechanism allows `app1` access to resources but denies `app2` access; this is circumvented by `app2` using `app1` as a facade to obtain access over a communication channel not monitored by the security mechanism. (b) A security mechanism denies `app1` access to resources; this is circumvented by accessing the resources through a side channel that bypasses the security mechanism.

being protected by the same permission. A classical example of a side channel attack is the timing attack to exfiltrate an encryption key from secure storage [42]. The system under attack is an algorithm that performs computation with the key and unintentionally leaks timing information—*i.e.*, how long it runs—that reveals critical information about the key.

Side channels are typically an unintentional consequence of a complicated system. ("Backdoors" are intentionally-created side channels that are meant to be obscure.) In Android, a large and complicated API results in the same data appearing in different locations, each governed by different access control mechanisms. When one API is protected with permissions, another unprotected method may be used to obtain the same data or an ersatz version of it.

2.3 App Analysis Methods

Researchers use two primary techniques to analyze app behaviour: static and dynamic analysis. In short, static analysis studies *software as data* by reading it; dynamic analysis studies *software as code* by running it. Both approaches have the goal of understanding the software's ultimate behaviour, but they offer insights with different certainty and granularity: static analysis reports instances of hypothetical behaviour; dynamic analysis gives reports of observed behaviour.

Static Analysis Static analysis involves scanning the code for all possible combinations of execution flows to understand potential execution behaviours—the behaviours of interest may include various privacy violations (*e.g.*, access to sensitive user data). Several studies have used static analysis to analyze different types of software in search of malicious behaviours and privacy leaks [4, 9–11, 19–22, 32, 37, 39, 41, 45, 92]. However, static analysis does not produce actual observations of privacy violations; it can only suggest that a violation may happen if a given part of the code gets executed at runtime. This means that static analysis provides an upper bound on hypothetical behaviours (*i.e.*, yielding false positives).

The biggest advantage of static analysis is that it is easy to perform automatically and at scale. Developers, however, have options to evade detection by static analysis because a program's runtime behaviour can differ enormously from its superficial appearance. For example, they can use code obfuscation [23, 29, 48] or alter the flow of the program to hide the way that the software operates in reality [23, 29, 48]. Native code in unmanaged languages allow pointer arithmetic that can skip over parts of functions that guarantee pre-conditions. Java's reflection feature allows the execution of dynamically created instructions and dynamically loaded code that similarly evades static analysis. Recent studies have shown that around 30% of apps render code dynamically [46], so static analysis may be insufficient in those cases.

From an app analysis perspective, static analysis lacks the contextual aspect, *i.e.*, it fails to observe the circumstances surrounding each observation of sensitive resource access and sharing, which is important in understanding when a given privacy violation is likely to happen. For these reasons, static analysis is useful, but is well complemented by dynamic analysis to augment or confirm findings.

Dynamic analysis Dynamic analysis studies an executable by running it and auditing its runtime behaviour. Typically, dynamic analysis benefits from running the executable in a controlled environment, such as an instrumented mobile OS [27, 85], to gain observations of an app's behaviour [16, 32, 46, 47, 50, 65, 66, 73, 85, 87–89].

There are several methods that can be used in dynamic analysis, one example is taint analysis [27, 32] which can be inefficient and prone to control flow attacks [68, 71]. A challenge to performing dynamic analysis is the logistical burden of performing it at scale. Analyzing a single Android app in isolation is straightforward, but scaling it to run automatically for tens of thousands of apps is not. Scaling dynamic analysis is facilitated with automated execution and creation of behavioural reports. This means that effective dynamic analysis

requires building an instrumentation framework for possible behaviours of interest *a priori* and then engineering a system to manage the endeavor.

Nevertheless, some apps are resistant to being audited when run in virtual or privileged environments [12, 68]. This has led to new auditing techniques that involve app execution on real phones, such as by forwarding traffic through a VPN in order to inspect network communications [44, 60, 63]. The limitations of this approach are the use of techniques robust to man-in-the-middle attacks [28, 31, 61] and scalability due to the need to actually run apps with user input.

A tool to automatically execute apps on the Android platform is the UI/Application Exerciser Monkey [6]. The Monkey is a UI fuzzer that generates synthetic user input, ensuring that some interaction occurs with the app being automatically tested. The Monkey has no context for its actions with the UI, however, so some important code paths may not be executed due to the random nature of its interactions with the app. As a result, this gives a lower bound for possible app behaviours, but unlike static analysis, it does not yield false positives.

Hybrid Analysis Static and dynamic analysis methods complement each other. In fact, some types of analysis benefit from a hybrid approach, in which combining both methods can increase the coverage, scalability, or visibility of the analyses. This is the case for malicious or deceptive apps that actively try to defeat one individual method (*e.g.*, by using obfuscation or techniques to detect virtualized environments or TLS interception). One approach would be to first carry out dynamic analysis to triage potential suspicious cases, based on collected observations, to be later examined thoroughly using static analysis. Another approach is to first carry out static analysis to identify interesting code branches that can then be instrumented for dynamic analysis to confirm the findings.

3   Testing Environment and Analysis Pipeline

Our instrumentation and processing pipeline, depicted and described in Figure 2, combines the advantages of both static and dynamic analysis techniques to triage suspicious apps and analyze their behaviours in depth. We used this testing environment to find evidence of covert- and side-channel usage in 252,864 versions of 88,113 different Android apps, all of them downloaded from the U.S. Google Play Store using a purpose-built Google Play scraper. We executed each app version individually on a physical mobile phone equipped with a customized operating system and network monitor. This testbed allows us to observe apps' runtime behaviours both at the OS and network levels. We can observe how apps request and access sensitive resources and their data sharing practices. We also have a comprehensive data analysis tool to de-obfuscate collected network data to uncover potential deceptive practices.
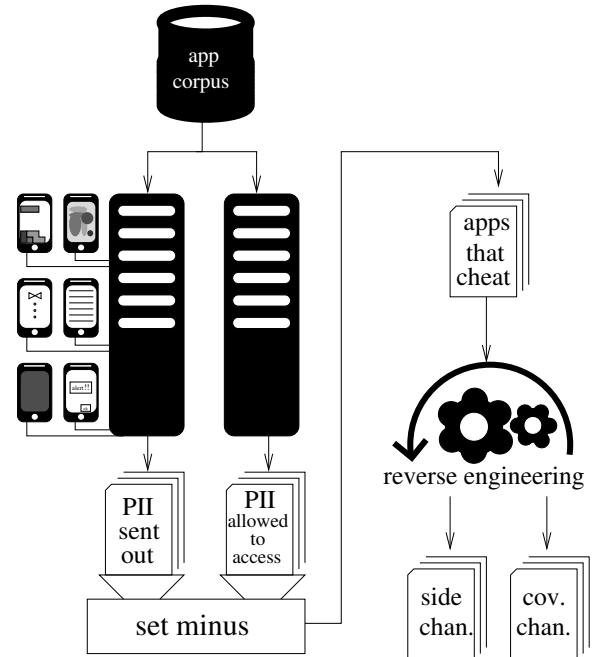

Figure 2: Overview of our analysis pipeline. Apps are automatically run and the transmissions of sensitive data are compared to what would be allowed. Those suspected of using a side or covert channel are manually reverse engineered.

Before running each app, we gather the permission-protected identifiers and data. We then execute each app while collecting all of its network traffic. We apply a suite of decodings to the traffic flows and search for the permission-protected data in the decoded traffic. We record all transmissions and later filter for those containing permission-protected data sent by apps not holding the requisite permissions. We hypothesize that these are due to the use of side and covert channels; that is, we are not looking for these channels, but rather looking for evidence of their use (*i.e.*, transmissions of protected data). Then, we group the suspect transmissions by the data type sent and the destination where it was sent, because we found that the same data-destination pair reflects the same underlying side or covert channel. We take one example per group and manually reverse engineer it to determine how the app gained permission-protected information without the corresponding permission.

Finally, we fingerprint the apps and libraries found using covert- and side-channels to identify the static presence of the same code in other apps in our corpus. A fingerprint is any string constant, such as specific filename or error message, that can be used to statically analyze our corpus to determine if the same technique exists in other apps that did not get triggered during our dynamic analysis phase.

3.1 App Collection

We wrote a Google Play Store scraper to download the most-popular apps under each category. Because the popularity distribution of apps is long tailed, our analysis of the 88,113 most-popular apps is likely to cover most of the apps that people currently use. This includes 1,505 non-free apps we purchased for another study [38]. We instrumented the scraper to inspect the Google Play Store to obtain application executables (APK files) and their associated metadata (*e.g.*, number of installs, category, developer information, etc.).

As developers tend to update their Android software to add new functionality or to patch bugs [64], these updates can also be used to introduce new side and covert channels. Therefore, it is important to examine different versions of the same app, because they may exhibit different behaviours. In order to do so, our scraper periodically checks if a new version of an already downloaded app is available and downloads it. This process allowed us to create a dataset consisting of 252,864 different versions of 88,113 Android apps.

3.2 Dynamic Analysis Environment

We implemented the dynamic testing environment described in Figure 2, which consists of about a dozen Nexus 5X Android phones running an instrumented version of the Android Marshmallow platform.[1] This purpose-built environment allows us to comprehensively monitor the behaviour of each of 88,113 Android apps at the kernel, Android-framework, and network traffic levels. We execute each app automatically using the Android Automator Monkey [6] to achieve scale by eliminating any human intervention. We store the resulting OS-execution logs and network traffic in a database for offline analysis, which we discuss in Section 3.3. The dynamic analysis is done by extending a platform that we have used in previous work [66].

Platform-Level Instrumentation   We built an instrumented version of the Android 6.0.1 platform (Marshmallow). The instrumentation monitored resource accesses and logged when apps were installed and executed. We ran apps one at a time and uninstalled them afterwards. Regardless of the obfuscation techniques apps use to disrupt static analysis, no app can avoid our instrumentation, since it executes in the system space of the Android framework. In a sense, our environment is a honeypot allowing apps to execute as their true selves. For the purposes of preparing our bug reports to Google for responsible disclosure of our findings, we retested our findings on a stock Pixel 2 running Android Pie—the most-recent version at the time—to demonstrate that they were still valid.

Kernel-Level Instrumentation   We built and integrated a custom Linux kernel into our testing environment to record apps' access to the file system. This module allowed us to record every time an app opened a file for reading or writing or unlinked a file. Because we instrumented the system calls to open files, our instrumentation logged both regular files and special files, such as device and interface files, and the `proc/` filesystem, as a result of the *"everything is a file"* UNIX philosophy. We also logged whenever an `ioctl` was issued to the file system. Some of the side channels for bypassing permission checking in the Android platform may involve directly accessing the kernel, and so kernel-level instrumentation provides clear evidence of these being used in practice.

We ignored the special device file `/dev/ashmem` (Android-specific implementation of asynchronous shared memory for inter-process communication) because it overwhelmed the logs due to its frequent use. As Android assigns a separate *user* (*i.e.*, uid) to each app, we could accurately attribute the access to such files to the responsible app.

Network-Level Monitoring   We monitored all network traffic, including TLS-secured flows, using a network monitoring tool developed for our previous research activities [63]. This network monitoring module leverages Android's VPN API to redirect all the device's network traffic through a localhost service that inspects all network traffic, regardless of the protocol used, through deep-packet inspection and in userspace. It reconstructs the network streams and ascribes them to the originating app by mapping the app owning the socket to the UID as reported by the `proc` filesystem. Furthermore, it also performs TLS interception by installing a root certificate in the system trusted certificate store. This technique allows it to decrypt TLS traffic unless the app performs advanced techniques, such as certificate pinning, which can be identified by monitoring TLS records and proxy exceptions [61].

Automatic App Execution   Since our analysis framework is based on dynamic analysis, apps must be executed so that our instrumentation can monitor their behaviours. In order to scale to hundreds of thousands of apps tested, we cannot rely on real user interaction with each app being tested. As such, we use Android's UI/Application Exerciser Monkey, a tool provided by Android's development SDK to automate and parallelize the execution of apps by simulating user inputs (*i.e.*, taps, swipes, etc.).

The Monkey injects a pseudo-random stream of simulated user input events into the app, *i.e.*, it is a UI fuzzer. We use the Monkey to interact with each version of each app for a period of ten minutes, during which the aforementioned tools log the app's execution as a result of the random UI events generated by the Monkey. Apps are rerun if the operation fails during execution. Each version of each app is run once in this manner; our system also reruns apps if there is unused capacity.

---

[1]While as of this writing Android Pie is the current release [35], Marshmallow and older versions were used by a majority of users at the time that we began data collection.

After running the app, the kernel, platform, and network logs are collected. The app is then uninstalled along with any other app that may have been installed through the process of automatic exploration. We do this with a white list of allowed apps; all other apps are uninstalled. The logs are then cleared and the device is ready to be used for the next test.

3.3 Personal Information in Network Flows

Detecting whether an app has legitimately accessed a given resource is straightforward: we compare its runtime behaviour with the permissions it had requested. Both users and researchers assess apps' privacy risks by examining their requested permissions. This presents an incomplete picture, however, because it only indicates what data an app *might* access, and says nothing about with whom it may share it and under what circumstances. The only way of answering these questions is by inspecting the apps' network traffic. However, identifying personal information inside network transmissions requires significant effort because apps and embedded third-party SDKs often use different encodings and obfuscation techniques to transmit data. Thus, it is a significant technical challenge to be able to de-obfuscate all network traffic and search it for personal information. This subsection discusses how we tackle these challenges in detail.

Personal Information   We define "personal information" as any piece of data that could potentially identify a specific individual and distinguish them from another. Online companies, such as mobile app developers and third-party advertising networks, want this type of information in order to track users across devices, websites, and apps, as this allows them to gather more insights about individual consumers and thus generate more revenue via targeted advertisements. For this reason, we are primarily interested in examining apps' access to the persistent identifiers that enable long-term tracking, as well as their geolocation information.

We focus our study on detecting apps using covert and side channels to access specific types of highly sensitive data, including persistent identifiers and geolocation information. Notably, the unauthorized collection of geolocation information in Android has been the subject of prior regulatory action [82]. Table 1 shows the different types of personal information that we look for in network transmissions, what each can be used for, the Android permission that protects it, and the subsection in this paper where we discuss findings that concern side and covert channels for accessing that type of data.

Decoding Obfuscations   In our previous work [66], we found instances of apps and third-party libraries (SDKs) using obfuscation techniques to transmit personal information over the network with varying degrees of sophistication. To identify and report such cases, we automated the decoding of a standard suite of standard HTTP encodings to identify personal information encoded in network flows, such as gzip, base64, and ASCII-encoded hexadecimal. Additionally, we search for personal information directly, as well as the MD5, SHA1, and SHA256 hashes of it.

After analyzing thousands of network traces, we still find new techniques SDKs and apps use to obfuscate and encrypt network transmissions. While we acknowledge their effort to protect users' data, the same techniques could be used to hide deceptive practices. In such cases, we use a combination of reverse engineering and static analysis to understand the precise technique. We frequently found a further use of AES encryption applied to the payload before sending it over the network, often with hard-coded AES keys.

A few libraries followed best practices by generating random AES session keys to encrypt the data and then encrypt the session key with a hard-coded RSA public key, sending both the encrypted data and encrypted session key together. To de-cipher their network transmissions, we instrumented the relevant Java libraries. We found two examples of third-party SDKs "encrypting" their data by XOR-ing a keyword over the data in a Viginère-style cipher. In one case, this was *in addition* to both using standard encryption for the data *and* using TLS in transmission. Other interesting approaches included reversing the string after encoding it in base64 (which we refer to as "46esab"), using base64 multiple times (base-base6464), and using a permuted-alphabet version of base64 (sa4b6e). Each new discovery is added to our suite of decodings and our entire dataset is then re-analyzed.

3.4 Finding Side and Covert Channels

Once we have examples of transmissions that suggest the permission system was violated (*i.e.*, data transmitted by an app that had not been granted the requisite permissions to do so), we then reverse engineer the app to determine how it circumvented the permissions system. Finally, we use static analysis to measure how prevalent this practice is among the rest of our corpus.

Reverse Engineering   After finding a set of apps exhibiting behaviour consistent with the existence of side and covert channels, we manually reverse engineered them. While the reverse engineering process is time consuming and not easily automated, it is necessary to determine how the app actually obtained information outside of the permission system. Because many of the transmissions are caused by the same SDK code, we only needed to reverse engineer each unique *circumvention technique*: not every app, but instead for a much smaller number of unique SDKs. The destination endpoint for the network traffic typically identifies the SDK responsible.

During the reverse engineering process, our first step was to use apktool [7] to decompile and extract the smali bytecode for each suspicious app. This allowed us to analyse and identify where any strings containing PII were created and from

Table 1: The types of personal information that we search for, the permissions protecting access to them, and the purpose for which they are generally collected. We also report the subsection in this paper where we report side and covert channels for accessing each type of data, if found, and the number of apps exploiting each. The dynamic column depicts the number of apps that we directly observed inappropriately accessing personal information, whereas the static column depicts the number of apps containing code that exploits the vulnerability (though we did not observe being executed during test runs).

| Data Type | Permission | Purpose/Use | Subsection | Nº of Apps | | Nº of SDKs | | Channel Type | |
				Dynamic	Static	Dynamic	Static	Covert	Side
IMEI	READ_PHONE_STATE	Persistent ID	4.1	13	159	2	2	2	0
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.2	42	12,408	1	1	0	1
Email	GET_ACCOUNTS	Persistent ID	Not Found						
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found						
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found						
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3	5	355	2	10	0	2
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found						
GPS	ACCESS_FINE_LOCATION	Location Data	4.4	1	1	0	0	0	1

which data sources. For some particular apps and libraries, our work also necessitated reverse engineering C++ code; we used IdaPro [1] for that purpose.

The typical process was to search the code for strings corresponding to destinations for the network transmissions and other aspects of the packets. This revealed where the data was already in memory, and then static analysis of the code revealed where that value first gets populated. As intentionally-obfuscated code is more complicated to reverse engineer, we also added logging statements for data and stack traces as new bytecode throughout the decompiled app, recompiled it, and ran it dynamically to get a sense of how it worked.

Measuring Prevalence The final step of our process was to determine the prevalence of the particular side or covert channel in practice. We used our reverse engineering analysis to craft a unique fingerprint that identifies the presence of an exploit in an embedded SDK, which is also robust against false positives. For example, a fingerprint is a string constant corresponding to a fixed encryption key used by one SDK, or the specific error message produced by another SDK if the operation fails.

We then decompiled all of the apps in our corpus and searched for the string in the resulting files. Within smali bytecode, we searched for the string in its entirety as a `const-string` instruction. For shared objects libraries like Unity, we use the `strings` command to output its printable strings. We include the path and name of the file as matching criteria to protect against false positives. The result is a set of all apps that may also exploit the side or covert channel in practice but for which our instrumentation did not flag for manual investigation, *e.g.*, because the app had been granted the required permission, the Monkey did not explore that particular code branch, etc.

4 Results

In this section, we present our results grouped by the type of permission that should be held to access the data; first we discuss covert and side channels enabling the access to persistent user or device IDs (particularly the IMEI and the device MAC address) and we conclude with channels used for accessing users' geolocation (*e.g.*, through network infrastructure or metadata present in multimedia content).

Our testing environment allowed us to identify five different types of side and covert channels in use among the 88,113 different Android apps in our dataset. Table 1 summarizes our findings and reports the number of apps and third-party SDKs that we find exploiting these vulnerabilities in our dynamic analysis and those in which our static analysis reveals code that *can* exploit these channels. Note that this latter category—those that *can* exploit these channels—were not seen as doing so by our instrumentation; this may be due to the Automator Monkey not triggering the code to exploit it or because the app had the required permission and therefore the transmission was not deemed suspicious.

4.1 IMEI

The International Mobile Equipment Identity (IMEI) is a numerical value that identifies mobile phones uniquely. The IMEI has many valid and legitimate operational uses to identify devices in a 3GPP network, including the detection and blockage of stolen phones.

The IMEI is also useful to online services as a persistent device identifier for tracking individual phones. The IMEI is a powerful identifier as it takes extraordinary efforts to change its value or even spoof it. In some jurisdictions, it is illegal to change the IMEI [56]. Collection of the IMEI by third parties facilitates tracking in cases where the owner tries to protect their privacy by resetting other identifiers, such as the advertising ID.

Android protects access to the phone's IMEI with the READ_PHONE_STATE permission. We identified two third-party online services that use different covert channels to access the IMEI when the app does not have the permission required to access the IMEI.

Salmonads and External Storage   Salmonads is a "third party developers' assistant platform in Greater China" that offers analytics and monetization services to app developers [67]. We identified network flows to salmonads.com coming from five mobile apps that contained the device's IMEI, despite the fact that the apps did not have permission to access it.

We studied one of these apps and confirmed that it contained the Salmonads SDK, and then studied the workings of the SDK closer. Our analysis revealed that the SDK exploits covert channels to read this information from the following hidden file on the SD card: /sdcard/.googlex9/.xamdecoq0962. If not present, this file is created by the Salmonads SDK. Then, whenever the user installs another app with the Salmonads SDK embedded and with legitimate access to the IMEI, the SDK—through the host app—reads and stores the IMEI in this file.

The covert channel is the apps' shared access to the SD card. Once the file is written, all other apps with the same SDK can simply read the file instead of obtaining access through the Android API, which is regulated by the permission system. Beyond the IMEI, Salmonads also stores the advertising ID—a resettable ID for advertising and analytics purposes that allows opting out of interest-based advertising and personalization—and the phone's MAC address, which is protected with the ACCESS_NETWORK_STATE permission. We modified the file to store new random values and observed that the SDK faithfully sent them onwards to Salmonads' domains. The collection of the advertising ID alongside other non-resettable persistent identifiers and data, such as the IMEI, undermines the privacy-preserving feature of the advertising ID, which is that it can be *reset*. It also may be a violation of Google's Terms of Service [36],

Our instrumentation allowed us to observe five different apps sending the IMEI without permission to Salmonads using this technique. Static analysis of our entire app corpus revealed that six apps contained the .xamdecoq0962 filename hardcoded in the SDK as a string. The sixth app had been granted the required permission to access the IMEI, which is why we did not initially identify it, and so it may be the app responsible for having initially written the IMEI to the file. Three of the apps were developed by the same company, according to Google Play metadata, while one of them has since been removed from Google Play. The lower bound on the number of times these apps were installed is 17.6 million, according to Google Play metadata.

Baidu and External Storage   Baidu is a large Chinese corporation whose services include, among many others, an online search engine, advertising, mapping services [14], and geocoding APIs [13]. We observed network flows containing the device IMEI from Disney's Hong Kong Disneyland park app (com.disney.hongkongdisneyland_goo) to Baidu domains. This app helps tourists to navigate through the Disney-themed park, and the app makes use of Baidu's Maps SDK. While Baidu Maps initially only offered maps of mainland China, Hong Kong, Macau and Taiwan, as of 2019, it now provides global services.

Baidu's SDK uses the same technique as Salmonads to circumvent Android's permission system and access the IMEI without permission. That is, it uses a shared file on the SD card so one Baidu-containing app with the right permission can store it for other Baidu-containing apps that do not have that permission. Specifically, Baidu uses the following file to store and share this data: /sdcard/backups/.SystemConfig/.cuid2. The file is a base64-encoded string that, when decoded, is an AES-encrypted JSON object that contains the IMEI as well as the MD5 hash of the concatenation of "com.baidu" and the phone's Android ID.

Baidu uses AES in CBC mode with a static key and the same static value for the initialization vector (IV). These values are, in hexadecimal, 33303231323130326469637564696162. The reason why this value is not superficially representative of a random hexadecimal string is because Baidu's key is computed from the binary representation of the ASCII string 30212102dicudiab—observe that when reversed, it reads as baidu cid 2012 12 03. As with Salmonads, we confirmed that we can change the (encrypted) contents of this file and the resulting identifiers were faithfully sent onwards to Baidu's servers.

We observed eight apps sending the IMEI of the device to Baidu without holding the requisite permissions, but found 153 different apps in our repository that have hardcoded the constant string corresponding to the encryption key. This includes two from Disney: one app each for their Hong Kong and Shanghai (com.disney.shanghaidisneyland_goo) theme parks. Out of that 153, the two most popular apps were Samsung's Health (com.sec.android.app.shealth) and Samsung's Browser (com.sec.android.app.sbrowser) apps, both with more than 500 million installations. There is a lower bound of 2.6 billion installations for the apps identified as containing Baidu's SDK. Of these 153 apps, all but 20 have the READ_PHONE_STATE permission. This means that they have legitimate access to the IMEI and can be the apps that actually create the file that stores this data. The 20 that do not have the permission can only get the IMEI through this covert channel. These 20 apps have a total lower bound of 700 million installations.

4.2 Network MAC Addresses

The Media Access Control Address (MAC address) is a 6-byte identifier that is uniquely assigned to the Network Interface Controller (NIC) for establishing link-layer communications. However, the MAC address is also useful to advertisers and analytics companies as a hardware-based persistent identifier, similar to the IMEI.

Android protects access to the device's MAC address with the ACCESS_NETWORK_STATE permission. Despite this, we observed apps transmitting the device's MAC address without having permission to access it. The apps and SDKs gain access to this information using C++ native code to invoke a number of unguarded UNIX system calls.

Unity and IOCTLs   Unity is a cross-platform game engine developed by Unity Technologies and heavily used by Android mobile games [77]. Our traffic analysis identified several Unity-based games sending the MD5 hash of the MAC address to Unity's servers and referring to it as a uuid in the transmission (*e.g.*, as an HTTP GET parameter key name). In this case, the access was happening inside of Unity's C++ native library. We reverse engineered libunity.so to determine how it was obtaining the MAC address.

Reversing Unity's 18 MiB compiled C++ library is more involved than Android's bytecode. Nevertheless, we were able to isolate where the data was being processed precisely because it hashes the MAC address with MD5. Unity provided its own unlabelled MD5 implementation that we found by searching for the constant numbers associated with MD5; in this case, the initial state constants.

Unity opens a network socket and uses an ioctl (UNIX "input-output control") to obtain the MAC address of the WiFi network interface. In effect, ioctls create a large suite of "numbered" API calls that are technically no different than well-named system calls like bind or close but used for infrequently used features. The behaviour of an ioctl depends on the specific "request" number. Specifically, Unity uses the SIOCGIFCONF[2] ioctl to get the network interfaces, and then uses the SIOCGIFHWADDR[3] ioctl to get the corresponding MAC address.

We observed that 42 apps were obtaining and sending to Unity servers the MAC address of the network card without holding the ACCESS_NETWORK_STATE permission. To quantify the prevalence of this technique in our corpus of Android apps, we fingerprinted this behaviour through an error string that references the ioctl code having just failed. This allowed us to find a total of 12,408 apps containing this error string, of which 748 apps do not hold the ACCESS_NETWORK_STATE permission.

---

[2] Socket ioctl get interface configuration
[3] Socket ioctl get interface hardware address

4.3 Router MAC Address

Access to the WiFi router MAC address (BSSID) is protected by the ACCESS_WIFI_STATE permission. In Section 2, we exemplified side channels with router MAC addresses being ersatz location data, and discussed the FTC enacting millions of dollars in fines for those engaged in the practice of using this data to deceptively infer users' locations. Android Nougat added a requirement that apps hold an additional location permission to scan for nearby WiFi networks [34]; Android Oreo further required a location permission to get the SSID and MAC address of the connected WiFi network. Additionally, knowing the MAC address of a router allows one to link different devices that share Internet access, which may reveal personal relations by their respective owners, or enable cross-device tracking.

Our analysis revealed two side channels to access the connected WiFi router information: reading the ARP cache and asking the router directly. We found no side channels that allowed for scanning of other WiFi networks. Note that this issue affects *all* apps running on recent Android versions, not just those without the ACCESS_WIFI_STATE permission. This is because it affects apps *without* a location permission, and it affects apps *with* a location permission that the user has not granted using the ask-on-first-use controls.

Reading the ARP Table   The Address Resolution Protocol (ARP) is a network protocol that allows discovering and mapping the MAC layer address associated with a given IP address. To improve network performance, the ARP protocol uses a cache that contains a historical list of ARP entries, *i.e.*, a historical list of IP addresses resolved to MAC address, including the IP address and the MAC address of the wireless router to which the device is connected (*i.e.*, its BSSID).

Reading the ARP cache is done by opening the pseudo file /proc/net/arp and processing its content. This file is not protected by any security mechanism, so any app can access and parse it to gain access to router-based geolocation information without holding a location permission. We built a working proof-of-concept app and tested it for Android Pie using an app that requests *no permissions*. We also demonstrated that when running an app that requests both the ACCESS_WIFI_STATE and ACCESS_COARSE_LOCATION permissions, when those permissions are denied, the app will access the data anyway. We responsibly disclosed our findings to Google in September, 2018.

We discovered this technique during dynamic analysis, when we observed one library using this method in practice: OpenX [57], a company that according to their website "creates programmatic marketplaces where premium publishers and app developers can best monetize their content by connecting with leading advertisers that value their audiences." OpenX's SDK code was not obfuscated and so we observed that they had named the responsible function

Table 2: SDKs seen sending router MAC addresses and also containing code to access the ARP cache. For reference, we report the number of apps and a lower bound of the total number of installations of those apps. We do this for all apps containing the SDK; those apps that *do not* have ACCESS_WIFI_STATE, which means that the side channel circumvents the permissions system; and those apps which *do* have a location permission, which means that the side channel circumvents location revocation.

SDK Name	Contact Domain	Incorporation Country	Total Prevalance		Wi-Fi Permission		No Location Permission	
			(Apps)	(Installs)	(Apps)	(Installs)	(Apps)	(Installs)
AIHelp	cs30.net	United States	30	334 million	3	210 million	12	195 million
Huq Industries	huq.io	United Kingdom	137	329 million	0	0	131	324 million
OpenX	openx.net	United States	42	1072 million	7	141 million	23	914 million
xiaomi	xiaomi.com	China	47	986 million	0	0	44	776 million
jiguang	jpush.cn	China	30	245 million	0	0	26	184 million
Peel	peel-prod.com	United States	5	306 million	0	0	4	206 million
Asurion	mysoluto.com	United States	14	2 million	0	0	14	2 million
Cheetah Mobile	cmcm.com	China	2	1001 million	0	0	2	1001 million
Mob	mob.com	China	13	97 million	0	0	6	81 million

getDeviceMacAddressFromArp. Furthermore, a close analysis of the code indicated that it would first *try* to get the data legitimately using the permission-protected Android API; this vulnerability is only used after the app has been explicitly denied access to this data.

OpenX did not directly send the MAC address, but rather the MD5 hash of it. Nevertheless, it is still trivial to compute a MAC address from its corresponding hash: they are vulnerable to a brute-force attack on hash functions because of the small number of MAC addresses (*i.e.*, an upper bound of 48 bits of entropy).[4] Moreover, insofar as the router's MAC address is used to resolve an app user's geolocation using a MAC-address-to-location mapping, one need only to hash the MAC addresses in this mapping (or store the hashes in the table) and match it to the received value to perform the lookup.

While OpenX was the only SDK that we observed exploiting this side channel, we searched our entire app corpus for the string /proc/net/arp, and found multiple third-party libraries that included it. In the case of one of them, igexin, there are existing reports of their predatory behaviour [15]. In our case, log files indicated that after igexin was denied permission to scan for WiFi, it read /system/xbin/ip, ran /system/bin/ifconfig, and then ran cat /proc/net/arp. Table 2 shows the prevalence of third-party libraries with code to access the ARP cache.

Router UPnP   One SDK in Table 2 includes another technique to get the MAC address of the WiFi access point: it uses UPnP/SSDP discovery protocols. Three of Peel's smart remote control apps (tv.peel.samsung.app, tv.peel.smartremote, and tv.peel.mobile.app) connected to 192.168.0.1, the IP address of the router that was their gateway to the Internet. The router in this configuration was a commodity home router that supports universal plug-and-play; the app requested the igd.xml (Internet gateway device configuration) file through

port 1900 on the router. The router replied with, among other manufacturing details, its MAC address as part of its UUID. These apps also sent WiFi MAC addresses to their own servers and a domain hosted by Amazon Web Services.

The fact that the router is providing this information to devices hosted in the home network is not a flaw with Android *per se*. Rather it is a consequence of considering every app on every phone connected to a WiFi network to be on the trusted side of the firewall.

4.4   Geolocation

So far our analysis has showed how apps circumvent the permission system to gain access to persistent identifiers and data that can be used to infer geolocation, but we also found suspicious behaviour surrounding a more sensitive data source, *i.e.*, the actual GPS coordinates of the device.

We identified 70 different apps sending location data to 45 different domains without having any of the location permissions. Most of these location transmissions were not caused by circumvention of the permissions system, however, but rather the location data was provided within incoming packets: ad mediation services provided the location data embedded within the ad link. When we retested the apps in a different location, however, the returned location was no longer as precise, and so we suspect that these ad mediators were using IP-based geolocation, though with a much higher degree of precision than is normally expected. One app explicitly used www.googleapis.com's IP-based geolocation and we found that the returned location was accurate to within a few meters; again, however, this accuracy did not replicate when we retested elsewhere [59]. We did, however, discover one genuine side channel through photo EXIF data.

Shutterfly and EXIF Metadata   We observed that the Shutterfly app (com.shutterfly) sends precise geolocation data to its own server (apcmobile.thislife.com) without holding a location permission. Instead, it sent photo metadata

---

[4]Using commodity hardware, the MD5 for every possible MAC address can be calculated in a matter of minutes [40].

from the photo library, which included the phone's precise location in its exchangeable image file format (EXIF) data. The app actually processed the image file: it parsed the EXIF metadata—including location—into a JSON object with labelled `latitude` and `longitude` fields and transmitted it to their server.

While this app may not be intending to circumvent the permission system, this technique can be exploited by a malicious actor to gain access to the user's location. Whenever a new picture is taken by the user with geolocation enabled, any app with read access to the photo library (*i.e.*, `READ_EXTERNAL_STORAGE`) can learn the user's precise location when said picture was taken. Furthermore, it also allows obtaining historical geolocation fixes with timestamps from the user, which could later be used to infer sensitive information about that user.

5 Related Work

We build on a vast literature in the field of covert- and side-channel attacks for Android. However, while prior studies generally only reported isolated instances of such attacks or approached the problem from a theoretical angle, our work combines static and dynamic analysis to automatically detect real-world instances of misbehaviours and attacks.

Covert Channels Marforio *et al.* [49] proposed several scenarios to transmit data between two Android apps, including the use of UNIX sockets and external storage as a shared buffer. In our work we see that the shared storage is indeed used in the wild. Other studies have focused on using mobile noises [26, 70] and vibrations generated by the phone (which could be inaudible to users) as covert channels [3, 17]. Such attacks typically involve two physical devices communicating between themselves. This is outside of the scope of our work, as we focus on device vulnerabilities that are being exploited by apps and third parties running in user space.

Side Channels Spreitzer *et al.* provided a good classification of mobile-specific side-channels present in the literature [74]. Previous work has demonstrated how unprivileged Android resources could be to used to infer personal information about mobile users, including unique identifiers [72] or gender [51]. Researchers also demonstrated that it may be possible to identify users' locations by monitoring the power consumption of their phones [52] and by sensing publicly available Android resources [91]. More recently, Zhang *et al.* demonstrated a sensor calibration fingerprinting attack that uses unprotected calibration data gathered from sensors like the accelerometer, gyroscope, and magnetometer [90]. Others have shown that unprotected system-wide information is enough to infer input text in gesture-based keyboards [72]. Research papers have also reported techniques that leverage

lowly protected network information to geolocate users at the network level [2, 54, 82]. We extend previous work by reporting third-party libraries and mobile applications that gain access to unique identifiers and location information in the wild by exploiting side and covert channels.

6 Discussion

Our work shows a number of side and covert channels that are being used by apps to circumvent the Android permissions system. The number of potential users impacted by these findings is in the hundreds of millions. In this section, we discuss how these issues are likely to defy users' reasonable expectations, and how these behaviours may constitute violations of various laws.

We note that these exploits may not necessarily be malicious and intentional. The Shutterfly app that extracts geolocation information from EXIF metadata may not be doing this to learn location information about the user or may not be using this data later for any purpose. On the other hand, cases where an app contains both code to access the data through the permission system and code that implements an evasion do not easily admit an innocent explanation. Even less so for those containing code to legitimately access the data and then store it for others to access. This is particularly bad because covert channels can be exploited by *any app* that knows the protocol, not just ones sharing the same SDK. The fact that Baidu writes user's IMEI to publicly accessible storage allows any app to access it without permission—not just other Baidu-containing apps.

6.1 Privacy Expectations

In the U.S., privacy practices are governed by the "notice and consent" framework: companies can give *notice* to consumers about their privacy practices (often in the form of a privacy policy), and consumers can *consent* to those practices by using the company's services. While website privacy policies are canonical examples of this framework in action, the permissions system in Android (or in any other platform) is another example of the notice and consent framework, because it fulfills two purposes: (i) providing transparency into the sensitive resources to which apps request access (notice), and (ii) requiring explicit user consent before an app can access, collect, and share sensitive resources and data (consent). That apps can and do circumvent the notice and consent framework is further evidence of the framework's failure. In practical terms, though, these app behaviours may directly lead to privacy violations because they are likely to defy consumers' expectations.

Nissenbaum's "Privacy as Contextual Integrity" framework defines privacy violations as data flows that defy contextual information norms [55]. In Nissenbaum's framework, data flows are modeled by senders, recipients, data subjects, data

types, and transmission principles in specific contexts (*e.g.*, providing app functionality, advertising, etc.). By circumventing the permissions system, apps are able to exfiltrate data to their own servers and even third parties in ways that are likely to defy users' expectations (and societal norms), particularly if it occurs after having just denied an app's explicit permission request. That is, regardless of context, were a user to explicitly be asked about granting an app access to personal information and then explicitly declining, it would be reasonable to expect that the data then would not be accessible to the app. Thus, the behaviours that we document in this paper constitute clear privacy violations. From a legal and policy perspective, these practices are likely to be considered deceptive or otherwise unlawful.

Both a recent CNIL decision (France's data protection authority), with respect to GDPR's notice and consent requirements, and various FTC cases, with respect to unfair and deceptive practices under U.S. federal law—both described in the next section—emphasize the notice function of the Android permissions system from a consumer expectations perspective. Moreover, these issues are also at the heart of a recent complaint brought by the Los Angeles County Attorney (LACA) under the California State Unfair Competition Law. The LACA complaint was brought against a popular mobile weather app on related grounds. The case further focuses on the permissions system's notice function, while noting that, "users have no reason to seek [geolocation data collection] information by combing through the app's lengthy [privacy policy], buried within which are opaque discussions of [the developer's] potential transmission of geolocation data to third parties and use for additional commercial purposes. Indeed, on information and belief, the vast majority of users do not read those sections at all" [76].

6.2 Legal and Policy Issues

The practices that we highlight in this paper also highlight several legal and policy issues. In the United States, for example, they may run afoul of the FTC's prohibitions against deceptive practices and/or state laws governing unfair business practices. In the European Union, they may constitute violations of the General Data Protection Regulation (GDPR).

The Federal Trade Commission (FTC), which is charged with protecting consumer interests, has brought a number of cases under Section 5 of the Federal Trade Commission (FTC) Act [79] in this context. The underlying complaints have stated that circumvention of Android permissions and collection of information absent users' consent or in a manner that is misleading is an unfair and deceptive act [84]. One case suggested that apps requesting permissions beyond what users expect or what are needed to operate the service were found to be "unreasonable" under the FTC Act. In another case, the FTC pursued a complaint under Section 5 alleging that a mobile device manufacturer, HTC, allowed developers

to collect information without obtaining users' permission via the Android permission system, and failed to protect users from potential third-party exploitation of a related security flaw [81]. Finally, the FTC has pursued cases involving consumer misrepresentations with respect to opt-out mechanisms from tailored advertising in mobile apps more generally [83].

Also in the United States, state-level Unfair and Deceptive Acts and Practices (UDAP) statutes may also apply. These typically reflect and complement the corresponding federal law. Finally, with growing regulatory and public attention to issues pertaining to data privacy and security, data collection that undermines users' expectations and their informed consent may also be in violation of various general privacy regulations, such as the Children's Online Privacy Protection Act (COPPA) [80], the recent California Privacy Protection Act (CCPA), and potentially data breach notification laws that focus on unauthorized collection, depending on the type of personal information collected.

In Europe, these practices may be in violation of GDPR. In a recent landmark ruling, the French data regulator, CNIL, levied a 50 million Euro fine for a breach of GDPR's transparency requirements, underscoring informed consent requirements concerning data collection for personalized ads [25]. This ruling also suggests that—in the context of GDPR's consent and transparency provisions—permission requests serve a key function of both informing users of data collection practices and as a mechanism for providing informed consent [81].

Our analysis brings to light novel permission circumvention methods in actual use by otherwise legitimate Android apps. These circumventions enable the collection of information either without asking for consent or after the user has explicitly refused to provide consent, likely undermining users' expectations and potentially violating key privacy and data protection requirements on a state, federal, and even global level. By uncovering these practices and making our data public,[5] we hope to provide sufficient data and tools for regulators to bring enforcement actions, industry to identify and fix problems before releasing apps, and allow consumers to make informed decisions about the apps that they use.

7 Limitations and Future Work

During the course of performing this research, we made certain design decisions that may impact the comprehensiveness and generalizability of this work. That is, all of the findings in this paper represent lower bounds on the number of covert and side channels that may exist in the wild.

Our study considers a subset of the permissions labeled by Google as *dangerous*: those that control access to user identifiers and geolocation information. According to Android's documentation, this is indeed the most concerning and privacy intrusive set of permissions. However, there may be

---

[5]https://search.appcensus.io/

other permissions that, while not labeled as *dangerous*, can still give access to sensitive user data. One example is the BLUETOOTH permission; it allows apps to discover nearby Bluetooth-enabled devices, which may be useful for consumer profiling, as well as physical and cross-device tracking. Additionally, we did not examine all of the dangerous permissions, specifically data guarded by content providers, such as address book contacts and SMS messages.

Our methods rely on observations of network transmissions that suggest the existence of such channels, rather than searching for them directly through static analysis. Because many apps and third-party libraries use obfuscation techniques to disguise their transmissions, there may be transmissions that our instrumentation does not flag as containing permission-protected information. Additionally, there may be channels that are exploited, but during our testing the apps did not transmit the accessed personal data. Furthermore, apps could be exposing channels, but never abuse them during our tests. Even though we would not report such behavior, this is still an unexpected breach of Android's security model.

Many popular apps also use certificate pinning [28, 61], which results in them rejecting the custom certificate used by our man-in-the-middle proxy; our system then allows apps to continue without interference. Certificate pinning is reasonable behaviour from a security standpoint; it is possible, however, that it is being used to thwart attempts to analyse and study the network traffic of a user's mobile phone.

Our dynamic analysis uses the Android Exerciser Monkey as a UI fuzzer to generate random UI events to interact with the apps. While in our prior work we found that the Monkey explored similar code branches as a human for 60% of the apps tested [66], it is likely that it still fails to explore some code branches that may exploit covert and side channels. For example, the Monkey fails to interact with apps that require users to interact with login screens or, more generally, require specific inputs to proceed. Such apps are consequently not as comprehensively tested as apps amenable to automated exploration. Future work should compare our approaches to more sophisticated tools for automated exploration, such as Moran et al.'s Crashscope [53], which generates inputs to an app designed to trigger crash events.

Ultimately, these limitations only result in the possibility that there are side and covert channels that we have not yet discovered (*i.e.*, false negatives). It has no impact on the validity of the channels that we did uncover (*i.e.*, there are no false positives) and improvements on our methodology can only result in the discovery of more of these channels.

Moving forward, there has to be a collective effort coming from all stakeholders to prevent apps from circumventing the permissions system. Google, to their credit, have announced that they are addressing many of the issues that we reported to them [33]. However, these fixes will only be available to users able to upgrade to Android Q—those with the means to own a newer smartphone. This, of course, positions privacy as a lux-ury good, which is in conflict with Google's public pronouncements [58]. Instead, they should treat privacy vulnerabilities with the same seriousness that they treat security vulnerabilities and issue hotfixes to all supported Android versions.

Regulators and platform providers need better tools to monitor app behaviour and hold app developers accountable by ensuring apps comply with applicable laws, namely by protecting users' privacy and respecting their data collection choices. Society should support more mechanisms, technical and other, that empower users' informed decision-making with greater transparency into what apps are doing on their devices. To this end, we have made the list of all apps that exploit or contain code to exploit the side and covert channels we discovered available online [8].

Acknowledgments

References

[1] IDA: About. Ida pro. https://www.hex-rays.com/products/ida/.

[2] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. WifiLeaks: Underestimated privacy implications of the access wifi state Android permission. Technical Report EURECOM+4302, Eurecom, 05 2014.

[3] A. Al-Haiqi, M. Ismail, and R. Nordin. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[5] Android Documentation. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2019. Accessed: February 12, 2019.

[6] Android Studio. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017. Accessed: October 12, 2017.

[7] Apktool. Apktool: A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[8] AppCensus Inc. Apps using Side and Covert Channels. https://appcensus.mobi/usenix2019, 2019.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI*, pages 259–269, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[12] G. S. Babil, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–8, July 2013.

[13] Baidu. Baidu Geocoding API. https://geocoder.readthedocs.io/providers/Baidu.html, 2019. Accessed: February 12, 2019.

[14] Baidu. Baidu Maps SDK. http://lbsyun.baidu.com/index.php?title=androidsdk, 2019. Accessed: February 12, 2019.

[15] Bauer, A. and Hebeisen, C. Igexin advertising network put user privacy at risk. https://blog.lookout.com/igexin-malicious-sdk, 2019. Accessed: February 12, 2019.

[16] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, San Diego, CA, 2014. USENIX Association.

[17] K. Block, S. Narain, and G. Noubir. An autonomic and permissionless android covert channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 184–194. ACM, 2017.

[18] S. Cabuk, C. E. Brodley, and C. Shields. IP covert channel detection. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):22, 2009.

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of NDSS*, 2015.

[20] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[21] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[22] M. Christodorescu, S. Jha, S. A Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[23] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2017.

[24] Commission Nationale de l'Informatique et des Libertés (CNIL). Data Protection Around the World. https://www.cnil.fr/en/data-protection-around-the-world, 2018. Accessed: September 23, 2018.

[25] Commission Nationale de l'Informatique et des Libertés (CNIL). The CNIL's restricted committee imposes a financial penalty of 50 Million euros against Google LLC, 2019.

[26] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *USENIX WOOT*, 2014.

[27] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[29] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 414–421. IEEE, 2014.

[30] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, page 3, New York, NY, USA, 2012. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[32] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Google, Inc. Android Q privacy: Changes to data and identifiers. https://developer.android.com/preview/privacy/data-identifiers#device-identifiers. Accessed: June 1, 2019.

[34] Google, Inc. Wi-Fi Scanning Overview. https://developer.android.com/guide/topics/connectivity/wifi-scan#wifi-scan-permissions. Accessed: June 1, 2019.

[35] Google, Inc. Distribution dashboard. https://developer.android.com/about/dashboards, May 7 2019. Accessed: June 1, 2019.

[36] Google Play. Usage of Google Advertising ID. https://play.google.com/about/monetization-ads/ads/ad-id/, 2019. Accessed: February 12, 2019.

[37] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[38] C. Han, I. Reyes, A. Elazari Bar On, J. Reardon, Á. Feal, S. Egelman, and N. Vallina-Rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. In *Workshop on Technology and Consumer Protection*, ConPro '19, 2019.

[39] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[40] Jeremi M. Gosney. Nvidia GTX 1080 Hashcat Benchmarks. https://gist.github.com/epixoip/6ee29d5d626bd8dfe671a2d8f188b77b, 2016. Accessed: June 1, 2019.

[41] J. Kim, Y. Yoon, K. Yi, J. Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[42] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[43] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[44] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, pages 15–20, 2015.

[45] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of ACM HotMobile*, page 2, 2012.

[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *badgers*, pages 3–17, 2014.

[47] M. Liu, H. Wang, Y. Guo, and J. Hong. Identifying and Analyzing the Privacy of Apps for Kids. In *Proc. of ACM HotMobile*, 2016.

[48] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[49] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[50] E. McReynolds, S. Hubbard, T. Lau, A. Saraf, M. Cakmak, and F. Roesner. Toys That Listen: A Study of Parents, Children, and Internet-Connected Toys. In *Proc. of ACM CHI*, 2017.

[51] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, pages 1053–1067, 2014.

[52] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, pages 785–800, 2015.

[53] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.

[54] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang. Unlocin: Unauthorized location inference on smartphones without being caught. In *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8. IEEE, 2013.

[55] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79:119, February 2004.

[56] United Kingdom of Great Britain and Northern Ireland. Mobile telephones (re-programming) act. http://www.legislation.gov.uk/ukpga/2002/31/introduction, 2002.

[57] OpenX. Why we exist. https://www.openx.com/company/, 2019.

[58] Sundar Pichai. Privacy Should Not Be a Luxury Good. The New York Times, May 7 2019. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html.

[59] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[60] A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 65–66. ACM, 2012.

[61] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362. ACM, 2017.

[62] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[63] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, 2015.

[64] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug fixes, improvements,... and privacy leaks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[65] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In *Proceedings of the ACM SIGMOBILE MobiSys*, pages 361–374, 2016.

[66] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. *Proceedings on Privacy Enhancing Technologies*, 2018(3):63–83, 2018.

[67] Salmonads. About us. http://publisher.salmonads.com, 2016.

[68] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[69] Sarah Schafer. With capital in panic, pizza deliveries soar. The Washington Post, December 19 1998. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm.

[70] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[71] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[72] L. Simon, W. Xu, and R. Anderson. Don't interrupt me while i type: Inferring text entered through gesture typing on android keyboards. *Proceedings on Privacy Enhancing Technologies*, 2016(3):136–154, 2016.

[73] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of ACM SPSM*, 2015.

[74] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[75] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. https://www.statista.com/statistics/266136, 2019. Accessed: February 11, 2019.

[76] COUNTY OF LOS ANGELES SUPERIOR COURT OF THE STATE OF CALIFORNIA. Complaint for injunctive relief and civil penalties for violations of the unfair competition law. http://src.bna.com/EqH, 2019.

[77] Unity Technologies. Unity 3d. https://unity3d.com, 2019.

[78] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.W. Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 145–162. USENIX Association, 2017.

[79] U.S. Federal Trade Commission. The federal trade commission act. (ftc act). https://www.ftc.gov/enforcement/statutes/federal-trade-commission-act.

[80] U.S. Federal Trade Commission. Children's online privacy protection rule ("coppa"). https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule, November 1999.

[81] U.S. Federal Trade Commission. In the Matter of HTC America, Inc. https://www.ftc.gov/sites/default/files/documents/cases/2013/07/130702htcdo.pdf, 2013.

[82] U.S. Federal Trade Commission. Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked, June 22 2016.

[83] U.S. Federal Trade Commission. In the Matter of Turn Inc. https://www.ftc.gov/system/files/documents/cases/152_3099_c4612_turn_complaint.pdf, 2017.

[84] U.S. Federal Trade Commission. Mobile security updates: Understanding the issues. https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf, 2018.

[85] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[86] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, May 2017.

[87] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[88] B. Yankson, F. Iqbal, and P.C.K. Hung. Privacy preservation framework for smart connected toys. In *Computing in Smart Toys*, pages 149–164. Springer, 2017.

[89] S. Yong, D. Lindskog, R. Ruhl, and P. Zavarsky. Risk Mitigation Strategies for Mobile Wi-Fi Robot Toys from Online Pedophiles. In *Proc. of IEEE SocialCom*, pages 1220–1223. IEEE, 2011.

[90] J. Zhang, A. R. Beresford, and I. Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[91] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM, 2013.

[92] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. Automated analysis of privacy requirements for mobile apps. In *24th Network & Distributed System Security Symposium*, 2017.